CRAY

RESEARCH INC.

Vectorization and Conversion of Fortran Programs for the CRAY-1 (CFT) Compiler
by
Lee Higbie
2240207

PREFACE

This technical note presents a number of techniques for promoting
vectorization in FORTRAN programs to be run on the Cray Research
CRAY-1 Computer System. Because the CRAY-1 FORTRAN Compiler (CFT) is
continually being refined, this note will be updated periodically.
With each update, I hope to increase the number of examples and expand
it in a few other ways to improve its usefulness.

I welcome any material that might be included in future editions, such
as more examples and coding techniques. I especially solicit help
with Appendix B where many errors of both omission and comission
undoubtedly lie. In particular, the various table positions that are
blank indicate that I don't know the proper entry. Special thanks are
due to the several people who sent me suggestions and examples. In
particular, a great deal of help for this revision was provided by
Dick Hendrickson.

<div align="right">LCH</div>

CONTENTS

SECTION 1

INTRODUCTION

This note describes techniques for helping to vectorize codes written for the CRAY-1 Computer and the CRAY-1 FORTRAN, CFT, Compiler System. Since the CFT Compiler is continually being refined, some of these techniques will become unnecessary. It is primarily intended to aid programmers vectorizing existing codes but should aid many programmers who are writing new codes to generate vectorizable loops.

Before going further, a caveat is in order. This note presents techniques for enhancing the vectorizability of codes only; it addresses few other methods of increasing program speed. Algorithm selection is generally far more important than coding techniques. For example, the FFT and various good sorting algorithms are approximately N/logN times as fast as the typical simplistic algorithms to perform the same tasks (for N input data), whereas the vectorization usually increases speed by a factor of 3 to 6. Thus, for typical dataset sizes, these best algorithms or other nearly optimal ones, are orders of magnitude faster than poor algorithms. No fancy coding techniques can overcome the use of ill chosen algorithms in such cases. Indeed, a good algorithm poorly coded is usually preferable to a poor one optimally coded.

Also, this note does not address to any great extent good programming practices which for CFT include (1) using few loops with long code blocks in preference to many short code loops; (2) judicious use of typing of variables; (3) long loops inside short loops rather than vice versa; and (4) if you are trying to get the last little bit from a vectorized loop, inserting extra parentheses starting at the end of an expression so that operations occur in an order that increases chaining. The techniques that are described are presented in six groups comprising the remaining sections of this note.

Sections 2 and 3 present the central issues that must be resolved before any useful work can be done, namely (1) finding the time consuming portions of the program and (2) circumventing overly modular or structured programming techniques. These tasks are so fundamental that compiler improvements are unlikely to be of much aid in the forseeable future without programmer help in these areas.

Section 4 discusses recursion, feedback, or vector dependency and introduces a simple but very useful and powerful technique; using directives which allow the programmer to indicate to the compiler that an individual block of code is logically vectorizable. Frequently, although the programmer knows from the physics of a situation that the code is vectorizable, coding in a form that allows the compiler to see this may not be convenient. Directives provide a means of pointing out vectorizable code to the compiler when this happens.

Section 5 shows one way to partially vectorize codes with irregular addressing—another anathema of vectorization, but one that the compiler will work around before too long.

Section 6 is a pot pourri of "tricks" to improve vectorization that do not readily fit into any of the earlier discussions.

The final section describes removing IF statements, a syntactic construct that the compiler will soon handle, at least in some cases.

Thus, Section 5 and 7 should be of less interest to programmers who are not in a big hurry to get the highest speed from their codes.

As a general note, CFT vectorizes innermost DO-loops only; it does not vectorize IF loops. Table 1 lists typical snytactic elements that may inhibit vectorization. Except for I/O statements, which often can be moved outside of loops after the debug phase, I discuss the more difficult of these constructs and how the programmer can remove these so that CFT will vectorize loops. Although not vectorized in the usual sense, unformatted I/O statements which involve arrays are processed with vector techniques.

CFT 1.06, The July 1979 release of CFT, vectorizes loops with constructs in the easy group (table 1) and allows scalar temporaries and user-provided (but CAL) functions from the second group. However, it inhibits vectorization for a loop containing other constructs in the second group or constructs in the third or fourth group. The third group includes constructs that are theoretically vectorizable but present challenges to the compiler writers. The items in the fourth group present a theoretical impossibility so that the only real hope for vectorizing loops containing them in the near future is to break the loop into several loops with the "impossible" construct in a separate (scalar) loop. If loops can be recast so that the inner DO loops include only items in the first categories, the chance of vectorization is enhanced and such loops that do not vectorize with CFT 1.06 are more likely to vectorize in the near future.

Table 1. FORTRAN inner DO-loop constructs

Difficulty	Syntactic constructs
Easy	- Long or complicated loops - Non unit incrementing of subscripts - Expressions in subscript - Intrinsic function references
Straightforward	- Scalar temporary variables - Function calls to programmer-supplied functions - Inner products - Logical IF statements - Transfer out of a loop (search loop) - Reduction operations
Difficult	- Linear recursion - IF statements - Some I/O - Complicated subscript expressions
"Impossible"	- Nonlinear indexing - Complicated branching within a loop - Ambiguous subscripting - Transfers into a loop - Subroutine calls - Nonlinear recursion - Some I/O

FINDING THE CENTRAL PORTION OF THE PROGRAM

Before spending your time vctorizing parts of the program that do not significantly affect the run time, first analyze the program to determine where it spends its time. This information may be readily available if the code is simple or if there is someone available who is familiar with it. Suppose, however, that this is the first time you've been faced with this task and that you have never seen the program before.

A typical, well behaved or "nice" program has a structure similar to that illustrated at right. With any luck, you will be able to find a similar pattern in your program and will be able to concentrate on the inner points of the program where your efforts will significantly impact the program's run time.

INITIALIZATION

BOUNDARY POINTS

INNER POINTS

DONE? AND I/0

NEXT CASE?
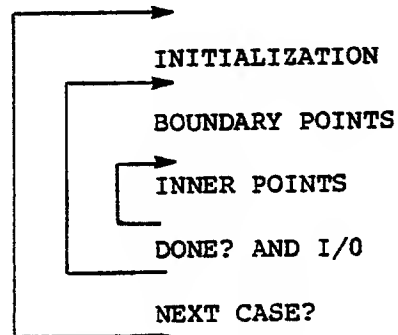
If you question the worth of this, look at a typical program and you are likely to see many simple vector loops that have been there all along. The trouble is, they intialize the grid and are not used for any of the computations! Since a problem with a grid of 100 points on a side has about 400 boundary points and about 10,000 interior points, working on the interior points and ignoring the boundary points -- let alone the initialization -- is clearly worthwhile. Even entirely removing the code for the boundary points leaves more than 96% of the points in the grid.

If you cannot discern the general structure, a worthwhile procedure is to use the flow analysis option in CFT to get a complete list of the subroutine calling tree and the time spent in each of the called routines. These figures tell you which routines consume the largest fractions of the time, thus it tells which routines are worth looking at. Refer to CFT Reference Manual, section 5, for a description of the Flowtrace option.

The flowtrace option adds a substantial overhead to every subroutine call and its output is lost if a job fails or executes a CALL EXIT.* Thus, if you have a program with many small subroutines, it is worthwhile to flowtrace a small case, at least for starters. You might also put a test such as the following in your code to stop the job after a reasonable length of time:

        IF(SECOND().GT.50)STOP**

To use the flowtrace option (CFT Manual section 5.4.5), put ON=F on the CFT statement:

        CFT,ON=F,... .

At the end of the run, you will get a table listing the time, percent of total time, number of times entered, and average time for each routine that is called as well as what routines called it and what routines it called. Only calls to FORTRAN programs that are compiled by the CFT,ON=F... statement are monitored;, $FTLIB, $SCILIB, $SYSLIB, and CAL routines are not monitored nor are the FORTRAN routines compiled separately without flowtrace enabled. Because of the great difference in execution speed of vectorized code compared to non-vectorized code, use of flowtrace is recommended even if you are familiar with a program. The timing analysis of flowtrace is frequently surprising.

---

*EXIT might be the name of one of your routines. Thus, the system cannot automatically assume that EXIT terminates a program. If you use EXIT to terminate your program, you can still use flowtrace by inserting the following subprogram in your deck.

        SUBROUTINE EXIT
        STOP 'EXIT'
        END

**This is one of many examples of non-standard FORTRAN employed in this note. CFT accepts all FORTRAN shown in the examples here.

GETTING AROUND OVERLY MODULAR OR

STRUCTURED PROGRAMS

Assume your code looks like this:

```
        DO 31    I = 2,99
        DO 31    J = 2,99
        CENTPT   = DATA(I,J)
        PTLEFT   = DATA(I-1,J)
        PTRGHT   = DATA(I+1,J)
        TEMP     = TEMPURTR(I,J)
        TEMPRT   = TEMPURTR(I+1,J)
        TEMPLFT  = TEMPURTR(I-1,J)
        CALL  INTGRTE(CENTPT,PTLEFT,PTRGHT,TEMPRT,TEMPLFT)
        CALL  EQNOST  (CENTPT,TEMP)
        DATA(I,J) = CENTPT
        TEMPURTR(I,J)= TEMP
31      CONTINUE
```

Your first impulse probably is to put this away until there are global
FORTRAN compiler that vectorize messes like this. However, this
represents a very common situation and is not nearly as hopeless as it
first appears. However it is hopeless for CFT thus, it is your chore
to put the DO loops inside the subroutine or, conversely, the
subroutines inside the DO loops. Putting DO loops inside subroutines
or the converse operation is probably the most complex part of
vectorizing codes and is the part that is most likely to be beneficial
in the long run. The other techniques discussed in this note are more
likely to be handled by the compiler or by a vectorizer someday.

Putting DO loops inside subroutines probably entails subscripting the
variable names being passed to the subroutines and passing the entire
arrays at once. Putting the subroutine in the loop means expanding
the subroutine code in line in the loop.

In the above loops, this can be done easily. Perhaps in your problem
the surface is not flat but is a sphere and so the right and left
points wrap around at the ends causing nonlinear indexing. Then, you
will have to try to separate the "bad points" and perhaps use some of
the techniques suggested in later sections.

To illustrate putting DO loops into subroutines and vice versa, suppose the subroutines are as follows:

```
SUBROUTINE INTGRTE(C,PL,PR,TL,TR)
COMMON DELTAX,DELTAT,GAMMAI,V,R
C = C + DELTAT *0.5 * (PL+PR) *DELTAX/(TR-TL)
RETURN
END


SUBROUTINE EQNOST(P,T,R)
COMMON DELTAX,DELTAT, GAMMAI,V,R
TP = (P/(V*R))**GAMMAI
RETURN
END
```

Then the two rewrites of the loop look like this:

Case 1. Putting loops inside Subroutines.

The entire DO 31 loop pair replaced by:

```
CALL  INTGRTV (DATA,TEMPURTR)

CALL  EQNOSTV (DATA, TEMPURTR)
```

Where INTGRTV is a vector version of INTRTE and EQNOSTV is a vectorized EQNOST.

Then, these new subroutines are:*

```
SUBROUTINE INTGRTV(D,T)
DIMENSION D(100,100), T(100,100)
COMMON DELTAX, DELTAT, GAMMAIV, R
DO  32  I=2,99
DO  32  J=2,99
D (I,J) = D(I,J) + DELTAT*0.5*(D(I-1,J)+D(I+1,J))
     *DELTAX/(T(I+1,J) - T(I-1,J))
32      CONTINUE
RETURN
END
```

---

*Reversing the order of the I and J loops would cause an unvectorizable dependency; see section 4.

```
            SUBROUTINE EQNOSTV (P,T)
            DIMENSION P(100,100), T(100,100)
            DO  33  I=2,99
            DO  33  J=2,99
            T(I,J) = (P(I,J)/(V*R))**GAMMAI
  33  ,     CONTINUE
            RETURN
            END
```

Case 2: Putting the subroutines inside the loops.

In this case, the DO 31 pair of loop becomes

```
            DO  34  I=2,99
            DO  34  J=2,99
            DATA (I,J) = DATA (I,J) + DELTAT *0.5* (DATA(I+1,J)
       $    +DATA(I-1,J))*DELTAX/(TEMPURTR(I+1,J)
            TEMPURTR(I,J) = (DATA(I,J)/(V*R))**GAMMAI


  34        CONTINUE
            RETURN
            END
```

The second alternative is also especially suitable for functions,
i.e., expand the code in line as in the next example:

```
            DO 35 I = 1,1000           ⎱    DO 36  I = 1,1000
  35        X(I) = ALOG2(Y(I)+1)...    ⎰36  X(I) = CST*ALOG(Y(I)+1)...
                      •
                      •
                      •
            FUNCTION ALOG2 (X)
            DATA CST /.../
            ALOG2 = CST*ALOG(X)
            RETURN
            END
```

The DO 35 loop will not vectorize because of the call to a routine
that the compiler doesn't recognize but the DO 36 loop will vectorize.

One common coding technique is to use vector subroutines such as VADD,
VMULT, and so on. The principal part of the program may then look
like this:

.

.

.

```
        CALL    VADD(A,B,C,N)

        CALL    VMULT(C,A,E,N)

        CALL    VADD(E,B,A,N)
```

.

.

.

Expanding these subroutines in line and, where possible, combining the
many DO loops into a few will ensure vectorization and will allow
intermediate variables to be held in registers rather than being
returned to memory:

```
        DO  37  I = 1,N

        A(I) = (B(I) + A(I)) * A(I) + B(I)

37      CONTINUE
```

Presumably, the VADD, VMULT, etc. vectorize but the DO 37 loop is
faster because the sum $A + B$ and the product $(A + B) * A$ do not have
to be stored, but can be kept in a register and A does not have to be
fetched a second time. Thus, the DO 34 loop is significantly faster
than the series of calls.

SECTION 4

RECURSION AND DIRECTING THE

COMPILER TO VECTORIZE

Suppose the key inner loop in your program is like the DO 41 inner loop, which doesn't vectorize and is therefore one that you want to spend some time on.

```
DO 41   I = 1,100

A(I) = A(I+L) ...
```

41   CONTINUE

If the loop is truly recursive*, the situation may be hopeless. However, if the value of L is such that there is no recursion (e.g., if L is greater than 1000), the easiest approach is to try directing the compiler to vectorize the loop and see if the answers remain the same. Placing the following compiler directive in front of the DO loop to be vectorized allows the compiler to vectorize a loop that has an apparent vector dependency or recursion:

        CDIR$ IVDEP   (see CFT manual sections 5.4, 5.4.3)

In other words, if either real or imagined recursion causes the loop not to be automatically vectorized by the compiler, the IVDEP compiler directive causes the computations to be done in vector mode. Note, however, that if CALL or IF statements or anything besides or in addition to apparent recursion prevents vectorization, CDIR$ IVDEP has no effect. Also, the effect of the IVDEP is limited to only the next DO loop; a separate IVDEP must be provided for each loop with an ignorable dependency; and the IVDEP should immediately precede the DO statement.

Returning to the example, first try printing some of the A terms the first few times through the vectorized loop to assure that vectorizing the loop does not change the results. Though this hardly proves that no problems can arise, it may help your analysis. This brute force approach is inelegant and error-prone, especially in those cases where one's insight into the physics of the situation does not provide some assurance that the loop is recursion-free. If the value of L differs with each pass through the loop, you may find it useful to make a copy of the loop with the compiler directive to vectorize, ignoring vector dependencies and a copy of the loop without the directive and select for use the vectorizable block only when it is correct. This means that you need to know what values of L are acceptable for vectorization of which loops.

---

*Recursion is a buzzword used to describe the case where output is propagated back into the input. It is explained further below.

Some rules of thumb to follow are the following:

1. If the sign of L is the same as the sign of the loop increment, there is never any recursion.

2. If the sign of L is the opposite of the loop index, there is probably recursion. An exception is when the loop increment and L have a least common multiple larger than the maximum value of the loop index.

3. There is no recursion of concern if the magnitude of L is such that there is no overlap of subscripts between the right and left sides of the computation. In fact, if L divided by the loop increment is greater than 64, you have no worries because the compiler breaks loops into 64-at-a-time blocks for vectorization.

If these simple rules do not help, you may have to analyze the problem further to determine when you can safely use vector computations.

"Recursion" is a mathematical term used to describe feedback, a noun you may find more familiar and easier to remember. The phenomenon referred to is the use of the output of one pass through the loop for the input to a computation on a subsequent pass. Consider the following simple examples:

```
        DO  42 I = 1,1000           SUM = 0
42        X(I) = X(I - 1) + 1        DO 43 I = 1,1000

                              43    SUM = SUM + X(I) * Y(I)
```

In the code on the left, the value of X(1) is used to compute X(2); X(2) is used to compute X(3), and so on. If this were done in vector mode, all of the X terms would be fetched at once, 1 would be added to each of them, and only the first value would be known to be correct. In the code on the right, the value of SUM is used for each subsequent pass through the loop. Inserting a CDIR$ IVDEP would probably produce wrong answers in the DO 42 loop and would have no affect on the DO 43 loop because the reason for scalar mode of the DO 43 loop is loop collapsing, as well as recursion.

The following loops are similar to these but are nonrecursive:

```
        DO 44 I = 1,1000           DO 45 I = 1,1000
44        X(I) = X(I + 1) + 1     45   A(I) = A(I) + X(I) * Y(I)
```

In the DO 44 loop, no X value is reused after being computed so there is no feedback. Similarly, the DO 45 loop is not recursive because no A value is reused after being generated; it is merely stored.

To understand the effects of recursion on vectorization, it is important to realize that vectorization is an essentially parallel computation on a group of values. Consider the simple case:

```
     DO 46 I = 2,3
     A(I-1) = 3.0
46   B (I)  = A(I)
```

This loop is equivalent to the sequential statements.

```
     A(1)  = 3.0
     B(2)  = A(2)
     A(2)  = 3.0
     B(3)  = A(3)
```

Vectorization, in effect, reorders the sequence to:

```
     A(1)  = 3.0
     A(2)  = 3.0
     B(2)  = A(2)    (Now 3.0)
     B(3)  = A(3)
```

and the "vectorized" sequence probably produces different results.

Whenever CFT encounters a loop which might be recursive, it generates correct scalar code rather than fast and possibly incorrect vector code, because vector and scalar versions of a recursive loop generally produce different results.

Recursion can cause problems if numerically equal subscript values occur on different passes through a DO loop and at least one of them is on the left of the equal sign.

There are two general classes of recursion:

1.   A value is prematurely destroyed if vectorized. The preceding is an example of this. The loop can often be made non-recursive by reordering the statements or by using temporary storage.

```
     DO 47  I = 2,3                DO 48 I = 2,3
     B (I)  = A (I)                TEMP    = A(I)
47   A(I-1) = 3.0                  A(I-1)  = 3.0
                              48   B(I)    = TEMP
```

2. A value is not ready when needed. This is the one-line recursion relationship:

```
        DO 49 I = 2,3
49      A(I) = B*A(I-1)+C
```

Because of the _group_ computation in vector mode, both input A values (the group A(1), A(2)) are used to compute the output value group A(2) and A(3). However, in this case the original A(2), not B*A(1)+C, is used to compute A(3), a probable error.

In many cases, CFT is not able to determine whether or not a subscript leads to recursion. For example:

```
        DO 410  I = 1,10
410     A(I,J) = A(I-1, JPLUS1)
```

is recursive if J is ever the same as JPLUS1. If the programmer knows from the physics of the situation, for example, that J and JPLUS1 will never be the same, then our

```
    CDIR$   IVDEP
```

is appropriate. Alternatively, the loop could be rewritten as

```
        DO 411 I = 1,3
411     A(I,J) = A(I-1,J+1)
```

and CFT would automatically vectorize it. In general, it is an aid to vectorization if subscripts can be explicity written out. For example:

```
        DO 412 I = 1,3
412     A(I) = A(I+N)
```

In many cases, N is not really a "variable"; it has a constant value and often never even changes from run to run. A "variable" is used simply to provide some flexibility in case the problem ever changes. Rather than initialize N with

```
        DATA N/3/
    or  N = 3
```

it is much better to use:

```
PARAMETER   (N = 3)
```

and CFT would automatically vectorize the sample loop.

In the following illustrations of recursive and non-recursive loops, assume that $X(I) = 2I$, $Y(I) = -I$, and $Z(I) = 0$ before the codes are run. The final values of X are given after the loop for subscripts = 1,2,3, ...

Recursive:

```
        DO 413   I = 2,5
        X(I) = X(I - 1) + 1.
413     CONTINUE

        X = 2, 3, 4, 5, 6
```

```
CDIR$  IVDEP
        DO 414 I = 2,5
        X(I) = X(I - 1) + 1.
414  CONTINUE
        X = 2, 3, 5, 7, 9
```

the last four of which are bad values because of forced vectorization of a recursive loop.

Non-recursive:

```
        DO 415   I  = 2,5
        X(I) = Y(I - 1) + 1.
415     CONTINUE
        X = 2, 0, -1, -2, -3
```

The compiler vectorizes the DO 415 loop automatically.

```
        DO 416 I = 1,5

        X(I) = X(I) + 1.
416     CONTINUE
        X = 3, 5, 7, 9, 11
```

The compiler vectorizes the DO 416 loop automatically.

```
        DO 417 I = 1,5

        X(I) = Z(I) + X(I) * Y(I)
417     CONTINUE
        X = -2, -8, -18, -32, -50
```

The compiler vectorizes the DO 417 loop automatically.

```
        L =  10                          L =  10
        DO 418 I = 1,5          CDIR$ IVDEP
        X(I) = X(I + L)                 DO 419 I = 1,5
418     CONTINUE                        X(I) = X(I + L)
        X = 22, 24, 26, 28, 30   419    CONTINUE
                                        X = 22, 24, 26, 28, 30
```

Here, the 418 loop does not vectorize but the 419 loop does. The
compiler does not know that L is not negative.

Recursive:

```
        L = -1                          L = -1
         .                               .
         .                               .
         .                               .
        DO 420 I = 1,5          CDIR$ IVDEP
        X(I) = X(I + L)                 DO 421 I = 1,5
420     CONTINUE                        X(I) = X(I + L)
        X = 0, 0, 0, 0, 0        421    CONTINUE
                                        X = 0, 2, 4, 6, 8

                                        The last four are incorrect
                                        because of forced vectoriza-
                                        tion of a recursive loop.
```

Here, the value of X(1) is fed back to compute X(2), i.e., the loop is
recursive and the vectorized version of the loop produces wrong
results. In scalar mode, the computations proceed...

```
        X(1) = X(0)       (=0 by assumption)
        X(2) = X(1)       (=0 from last computation)
        X(3) = X(2)       (=0 from last computation)
        X(4) = X(3)       (=0 from last computation)
        X(5) = X(4)       (=0 from last computation)
```

CFT generates code that executes as above because its approach to vectorization is conservative. When forced to vectorize, the loop executes:

X = shifted X = (0, 2, 4, 6, 8)

so that X(2) = original value of X(1), not the just-computed value; similarly, X(3) = original X(2), not the newly computed value, etc. Also, in this example it is assumed that 0 is a legal subscript, i.e., X is declared DIMENSION X(0:50).

Many examples of recursion are of the following form where L is negative (that is, opposite in sign to the increment of J, which is 1 here) and K is positive (of the same sign as the increment of I in this example):

```
      DO 422 I = 1,100
      DO 422 J = 1,100
422   A(I,J) = A(I + K, J + L) ...
```

Here, by inverting the order of the loops, you can remove the recursion and allow vectorization by using the CDIR$ IVDEP directive. This type of loop order inversion is frequently too complex to analyze easily and you may need to go back to the physics of the situation or to that unfortunate alternative of printing gobs of values to determine a reasonable way to reorder or rewrite the code.

The following examples show a simple but real case where the compiler's overly conservative attitude is easy to see and correct. Case 1 runs about four times slower than Case 2. The cause of such a large speed increase is the complexity of the loop. Loops with very few computations generally have less speed up.

CASE 1

```
          NL1 = 1
          NL2 = 2

             .

             .

             .

          DO 423 KX = 2,3
          DO 423 KY = 2,21
          DU1 = U1(KX,KY + 1,NL1) - U1(KX,KY - 1,NL1)
          DU2 = U2(KX,KY + 1,NL1) - U2(KX,KY - 1,NL1)
          DU3 = U3(KX,KY + 1,NL1) - U3(KX,KY - 1,NL1)
          U1(KX,KY,NL2) = U1(KX,KY,NL1)+A11*DU1+A12*DU2+A13*DU3
     $    +SIG*(U1(KX+1,KY,NL1)-2.*U1(KX,KY,NL1)+U1(KX-1,KY,NL1))
          U2(KX,KY,NL2) = U2(KX,KY,NL1)+A21*DU1+A22*DU2+A23*DU3
     $    +SIG*(U2(KX+1,KY,NL1)-2.*U2(KX,KY,NL1)+U2(KX-1,KY,NL1))
          U3(KX,KY,NL2) = U3(KX,KY,NL1)+A31*DU1+A32*DU2+A33*DU3
     $    +SIG*(U3(KX+1,KY,NL1)-2.*U3(KX,KY,NL1)+U3(KX-1,KY,NL1))
423       CONTINUE

             .

             .

             .
```

The values of NL1 and NL2 are swapped before the next pass through loop.

CASE 2

.

.

.

```
          DO 424 KX = 2,3
CDIR$ IVDEP
          DO 424 KY = 2,21
          DU1 = U1(KX,KY + 1,NL1) - U1(KX,KY - 1,NL1)
          DU2 = U2(KX,KY + 1,NL1) - U2(KX,KY - 1,NL1)
          DU3 = U3(KC,KY + 1,NL1) - U3(KX,KY - 1,NL1)
          U1(KX,KY,NL2) = U1(KX,KY,NL1)+A11*DU1+A12*DU2+A13*DU3
     $    +SIG*(U1(KX+1,KY,NL1)-2.*U1(KX,KY,NL1)+U1(KX-1,KY,NL1))
          U2(KX,KY,NL2) = U2(KX,KY,NL1)+A21*DU1+A22*DU2+A23*DU3
     $    +SIG*(U2(KX+1,KY,NL1)-2.*U2(KX,KY,NL1)+U2(KX-1,KY,NL1))
          U3(KX,KY,NL2) = U3(KX,KY,NL1)+A31*DU1+A32*DU2+A33*DU3
     $    +SIG*(U3(KX+1,KY,NL1)-2.*U3(KX,KY,NL1)+U3(KX-1,KY,NL1))
424       CONTINUE
```

.

.

.

I hope these examples shed some light on this rather abstruse topic.

IRREGULAR ADDRESSING

Irregular or nonlinear addressing arises in situations such as those using data structures requiring subscripted subscripts. Subscripted subscripts do not occur explicitly in FORTRAN-66 code but may effectively occur in certain types of programs as below:

In the DO 51 loop, Y essentially has a subscripted subscript

```
        DO 51 I = 1,1000

        J = INDEX (I)

        X(I) = Y(J) ...

51      CONTINUE
```

Change to:

```
        DO 52 I = 1,1000

        J = INDEX (I)

52      TEMP(I) = Y(J)

        DO 53 I = 1,1000

53      X(I) = TEMP(I) ...
```

The DO 51 loop cannot vectorize with CFT 1.06 because of the nonlinear indexing. The DO 52 loop similarly doesn't vctorize but the DO 53 loop does and, if the computations are extensive, the speed-up can be dramatic.

In general, if the computations are sufficiently complicated to warrant the work, you can restructure the loop into two or three loops. The first new loop is a GATHER loop in which all the data to be manipulated are collected into vectors. Next is the computation loop. Then is the SCATTER loop, in which results are distributed from the vector used in the computation loop to their proper locations. Quite often, as in this example, there is no SCATTER loop. There are $SCILIB routines for doing the GATHER and SCATTER (see Appendix A and CRI Manual 2240014).

The following example illustrates this again for a particle pushing algorithm.

CASE 1

```
        DO 54 K = 1,150
        IX = GRD(K)
        XI = IX
        VX(K) = VX(K) + EX(IX) + (XX(K) - XI) * DEX(IX)
        XX(K) = XX(K) + VX(K) + FLX
C       IX IS, IN EFFECT, A SUBSCRIPTED SUBSCRIPT
        IR = XX(K)
        RI = IR
        RX1 = XX(K) - RI
        IR = IR - (IR/64) * 64
        XX(K) = RI + RX1
C       IR IS AN IRREGULAR SUBSCRIPT
        RH(IR) = RH(IR) + 1.0 - RX1
        RH(IR + 1) = RH(IR + 1) + RX1
54      CONTINUE
```

CASE 2

```
          DO 55 K = 1,150
          IX = GRD(K)
          XIV(K) =  IX
          EXC(K) = EX(IX)
          DEXC(K) = DEX(IX)
55        CONTINUE
C         GATHER LOOP ABOVE
C         XI IS VECTORIZED INTO XIV
C         EX IS GATHERED INTO EXC
C         DEX IS GATHERED INTO DEXC
          DO 56 K = 1,150
          VX(K) = VX(K) + EXC(K) + (XX(K) - XIV(K)) * DEXC(K)
          XX(K) = XX(K) + VX(K) + FLX
          IRV(K) = XX(K)
          RI = IRV(K)
          RX1V(K) = XX(K) - RI
          XX(K) = RI + RX1V(K)
56        CONTINUE
C         COMPUTATION LOOP WHICH VECTORIZES IS ABOVE
          DO 57 K = 1,150
          RH(IRV(K)) = RH(IRV(K)) + 1.0 - RX1V(K)
          RH(IRV(K) + 1) = RH(IRV(K) + 1) + RX1V(K)
57        CONTINUE
C         SCATTER LOOP
```

The code in Case 2 runs more than twice as fast as that in Case 1.

SECTION 6

MISCELLANEOUS TECHNIQUES

This section includes a group of examples that do not readily fit into the categories discussed above. In some sense, this is a bag-of-tricks chapter demonstrating several additional loop restructuring techniques as well as all multi-loop techniques. The techniques here are harder to describe in a general and systematic fashion.

A matrix multiply represents an algorithm that can benfit from loop restructuring. For example, the following code illustrates the common way of coding the matrix multiply:

```
      DO 61 I = 1,L
      DO 61 J = 1,M
      C(I,J)  = 0.0
      DO 61 K = 1,N
61    C(I,J)  = C(I,J) + A(I,K) * B(K,J)
```

The recursion on C(I,J) and loop collapsing prevent vectorization now (CFT 1.06) and will always prevent as full vectorization as the rewrite below. This rewritten code vectorizes fully, resulting in a speedup of 5 to 10 times:

```
      DO 63 I = 1,L
      DO 62 J = 1,M
62    C(I,J)  = 0
      DO 63 K = 1,N
      DO 63 J = 1,M
63    C(I,J)  = C(I,J) + A(I,K) * B(K,J)
```

In many similar situations, although the result is not going into a subscripted variable but into a scalar temporary you can reorder the loops and store the results as a vector temporary instead of as a scalar temporary.

The next example shows several stages in the speed-up process. Case 2
is more than 50% faster than Case 1 and Case 3 is almost four times as
fast as Case 1.

CASE 1

```
        Q = 0.0
        DO 64 K = 1,996,5
        Q = Q + Z(K) * X(K) + Z(K + 1) * X(K + 1)
   $            + Z(K +2) * X (K + 2) + Z(K + 3) * X(K +3)
   $            + Z(K + 4) * X(K + 4)
64      CONTINUE
```

In this original case, the loop was quintupled, presumably to cut loop
overhead or allow greater overlap of operations.

CASE 2

```
        DO 65 K = 1,996,5
        TP(K) = Z(K) * X(K) + Z(K + 1) * X(K + 1)
   $            + Z(K + 2) * Z(K + 2) + Z(K + 3) * X(K + 3)
   $            + Z(K + 4) * X(K + 4)
65      CONTINUE
        Q = 0.0
        DO 66 K = 1,996,5
        Q = Q + TP(K)
66      CONTINUE
```

CASE 3

        Q = SDOT(1000,Z,1,X,1)

Here, SDOT is the BLA single-precision dot function (see Appendix A or
CRI publication 2240204).

As an aid to remembering the calling sequences for the basic linear
algebra functions, the first argument is the vector length, the
remaining arguments are in pairs: a vector operand followed by its
increment in memory.

Thus, if A and B are declared DIMENSION A(M,N),B(N,L) and you want to
compute the dot product of the Ith row of A with the Jth column of B,
use:

        AB = SDOT(N,A(I,1),M,B(1,J),1)

where N = the vector length = number of elements in each vector operand,
A(I,1) and B(1,J) are the starting locations in memory of the operands,
M = memory increment of the first operand vector and 1 = memory
increment of the second operand vector

Appendix A lists the BLA subroutines briefly as well as a few other
useful routines that are in $SCILIB.

A planned enhancement to CFT is to perform scalar operations for
individual statements in otherwise vectorizable loops. An industrious
programmer can achieve this now by using VFUNCTIONS:

        CDIR$ VFUNCTION ...

which tells the compiler of external non-libray vector functions. For
CFT 1.06, these can be written only in CAL. The CFT manual sections 5
and Appendix F provide the information necessary to link such routines
to FORTRAN programs.

SECTION 7

REMOVING IF STATEMENTS AND USING

BUILT-IN FUNCTIONS

CFT 1.06 does not vectorize code blocks that contain IF statements.
Many types of loops with IF statements are not hopeless, however.
Several things can be done depending on the structure of the code. CFT
will eventually vectorize many of these for you but in the meantime you
can help by using some of the built-in functions such as AMAX1, ABS,
CVMGT, CVMGZ, ...etc. (See CFT manual appendixes B and C). For example:

```
        DO 71 I = 1,1000
        IF(A(I) .LT. 0.) A(I) = 0.
71      B(I) = SQRT (A(I)) ...
```

which can be converted to:

```
        DO 72 I = 1,1000
        A(I) = AMAX1(A(I),0.)
72      B(I) = SQRT (A(I)) ...
```

The DO 71 loop doesn't vectorize now; the DO 72 loop does.

All the built-in arithmetic functions in FORTRAN (in $FTLIB) have both
vector and scalar versions; the compiler calls the vector version for
vectorizable loops*. The vector merge operations, CVMG*, are typeless
functions that allow you to merge the results of different vector
computations such as the following:

```
        DO 74 I = 1,1000
        IF(A(I) .LT. 0.) GOTO 73
        B(I) = A(I) + D(I) ...
        GOTO 74
73      B(I) = A(I) * E(I) ...
74      CONTINUE
```

This can be rewritten to vectorize as

```
        DO 75 I = 1,1000
75      B(I) = CVMGT (A(I) * E(I) ..., (A(I) + D(I) ..., A(I).LT.0)
```

---

*Some are actually pseudo vector routines; they allow the loop to
 vectorize but are performed in scalar mode.

The mnemonic for the CVMG* group of functions is that the last letter of the name is the condition on which the first argument is used. Since these functions are Boolean, they can be used with integer or floating operands and results and in scalar loops as well as in vector loops. Thus, if you are not sure that you are computing the value of B correctly, you can put a print statement in the loop, which causes it to be scalar, and still obtain the same results, albeit much slower than before.

Table 2 lists the merge functions and some of the other ones that you may want in similar situations.

Table 2. Some typical in line CFT functions.

FUNCTION NAME	RESULT TYPE	ARGUMENT TYPES	OPERATION
AMAX0 $(X_1,X_2...)$	Real	Real	Largest X_1
AMAX1 $(I_1,I_2...)$	Real	Integer	Largest I_1, floated
MAX0 $(X_1,X_2...)$	Integer	Real	Largest X_1, truncated
MAX1 $(I_1,I_2...)$	Integer	Integer	Largest I_1
CVMGT (X,Y,L)	Boolean (single word)	Boolean (single word)	X if L True, otherwise Y
CVMGZ (X,Y,Z)	Boolean (single word)	Boolean (single word)	X if Z is zero, Y if Z is nonzero

Another technique that works in some cases is inverting the order of loops so that the IF statements are in the outer loops rather than in the inner loops. Also, if the purpose of the IF test is to separate an exceptional case from other cases and if the computation is extensive, it may be worthwhile to write a loop to do the testing and to write a vectorizing loop for the computations.

Here are some more examples:

```
        Y(I) = 1.0
        IF(X(I).EQ.0.)  GOTO 76
        Y(I) = 1.0/X(I)
76      CONTINUE
```

Change this to:

```
        Y(I) = 1.0/CVMGZ (1.,X(I),X(I)) ...
```

which allows a loop containing it to vectorize and yet does not cause a divide fault. Here, CVMGZ selects 1. when X(I) = 0; otherwise it selects X(I). Alternatively, if this exceptional condition only occurs in cases when the result is not used, you can surround the loop containing it with CALL CLEARFI and CALL SETFI to turn the floating point interrupt off and then on again. This allows generation of an infinity without interrputing the program.

The next example illustrates loop reordering and IF statement removal:

```
      CASE 1                                 CASE 2

          DO 77 K = 1,3                          DO 710 JA = 1,500

          FR(K) = 0                              DSV(JA) = 0

77        CONTINUE                      C        DSV IS A VECTOR OF DS VALUES

          DO 79 JA = 1,500              710      CONTINUE

          IF (JA .EQ IA) GOTO 79                 DO 712 K = 1,3

          DS = 0                                 DO 711 JA = 1,500

          DO 78 K = 1,3                          AM(K,JA) = RS(K,IA) - RS(K,JA)

          A(K) = RS(K,IA) - RS(K,JA)   711      DSV(JA) + AM(K,JA) ** 2

          DS = DS + A(K) ** 2          712      CONTINUE

78        CONTINUE                              DO 713 JA = 1,500

          DS = SQRT(DS)                          DSV(JA) = SQRT(DSV(JA))

          IF (DS .GT. RAD) GOTO 79     713      CONTINUE

             .                                  DO 714 JA = 1,500

             .                                  IF (DSV(JA) .GT. RAD)   GOTO 714

             .                                     .

                                                   .

                                                   .

79        CONTINUE

                                       714      CONTINUE
```

Some of the loops in Case 1 vectorize but this vector length is only 3. In Case 2, the inner loops vectorize with a vector length of 500. In particular, the 713 loop uses a vector square root saving a great deal of time. As it turns out, in the "real life" example, most of the time DS was greater than RAD (last statement shown in the loop) so the rest of the loop did not need any work. Even though additional vectorization could be done, it would not have been very productive. With the change illustrated, the entire kernel ran more than four times faster than the original.

$SCILIB SUBROUTINES

This appendix summarizes the scientific library subroutines.
For a current and more complete description of these functions, refer to
the Library Subroutine Reference Manual, CRI Publication 2240014.

LEGEND:

N	Vector length
X,Y	Floating point vectors
IX,IY	Increments in memory of floating point vectors
C,D	Complex vectors
IC,ID	Increments in memory of complex vectors
NB	Number of bits per word selected for PACK/UNPACK
NW	Number of words in unpacked array.

Name(Parameters)	Type	Purpose
ISAMAX(N,X,IX)	Integer function	Index to real array element having maximum absolute value
ICAMAX(N,C,IC)	Integer function	Index to complex array element having maximum modulus.
SASUM(N,X,IX)	Real function	Sums the absolute value of a real array
SCASUM(N,C,IC)	Real function	Sums the absolute values of real and imaginary parts of complex array
SAXPY(N,X,IX,Y,IY)	Subroutine	Performs vector computations y←ax+y on real arrays, x,y.
CAXPY(N,C,IC,D,ID)	Subroutine	Performs vector computation y←ax+y in complex arrays x,y.
SCOPY(N,X,IX,Y,IY)	Subroutine	Copies real array x into real array y.
CCOPY(N,C,IC,D,ID)	Subroutine	Copies complex array c into complex array d.

Name (Parameters)	Type	Purpose
SDOT(N,X,IX,Y,IY)	Real function	DOT product of real arrays x,y.
CDOTC(N,C,IC,D,ID)	Complex function	DOT product of complex arrays c,d.
CDOTU(N,C,IC,D,ID)	Complex function	DOT product of complex arrays c,d.
SNRM2(N,X,IX,Y,IY)	Real function	Euclidean norm of real array x.
SCNRM2(N,C,IC)	Real function	Euclidean norm of complex array c.
SROT(N,X,IX,Y,IY)	Subroutine	Performs Givens transformation on real arrays x,y.
SROTG(...)	Subroutine	Calculates parameters for SROTX.
SROTM(...)	Subroutine	Modified Givens transformation.
SROTMG(...)	Subroutine	Sets up parameters for modified Givens.
SSCAL(N,A,X,IX)	Subroutine	Rescales real array x: x←ax, a real.
CSSCAL(N,A,C,IC)	Subroutine	Rescales complex array c: c←ac, with a real.
CSCAL(N,A,C,IC)	Subroutine	Rescales complex array c: c←ac, with a complex.
SSWAP(N,X,IX,Y,IY)	Subroutine	Swaps real arrays x,y.
CSWAP(N,C,IC,D,ID)	Subroutine	Swaps complex arrays c,d.
MXMA(...)	Subroutine	Completely general matrix multiply.
CFFT2(...)	Subroutine	Fourier transforms binary radix complex array.
RCFFT2(...)	Subroutine	Fourier transforms binary radix real to complex.
CRFFT2(...)	Subroutine	Fourier transforms binary radix complex array to real.

Name (Parameters)	Type	Purpose
PACK(P,NB,U,NW)	Subroutine	Packs power of 2 bit partial word lists.
UNPACK(P,NB,U,NW)	Subroutine	Unpacks list into power of 2 bit partial words.
MINV(...)	Subroutine	Returns solution of general linear equation set, matrix inverse optional.
SSUM(N,C,IC,D,ID)	Real function	Sums the elements of a real array.
CSUM(N,C,IC)	Complex function	Sums the elements of a complex array.
CROT(N,X,IX,Y,IY)	Subroutine	Applies complex Givens rotation.
CROTG(N,C,IC,D,ID)	Subroutine	Sets up rotational parameters for CROT.
FILTERG(...)	Subroutine	Performs general fitering and auto-correlation.
FILTERS(...)	Subroutine	Calculates symmetric filter coefficient.
OPFLIT(...)	Subroutine	Wiener-Levinson equation solver.

APPENDIX B

FORTRAN DIALECTICAL DIFFERENCES

As an aid for conversions, this appendix contains a number of tables that compare the FORTRAN compiler dialects for several manufacturers. The following tables are included.

1. Hardware dependencies

2. Coding features

3. Declaratives and ordering

4. Names and variables

5. Constants, literals, and strings

6. Arithmetic and expressions

7. Branching and control statements

8. Input/output formatting

9. Subroutines and functions

10. Instrinsic or inline functions

11. External functions.

All of these tables are based on a scan of manuals and are thus prone to error and very prone to omissions. Further, as manufacturers bring their FORTRAN dialects into conformance with FORTRAN X3.9-1978, some of these differences can be expected to disappear. The tables also reflect 1975-1979 versions of FORTRAN. In particular, while CDC FTN 5 is to be an ANSI-1978 version here. CDC means either FTN 4 or RUN FORTRAN. IBM means FORTRAN H. Univac means either FORTRAN V or ASCII FORTRAN. ICL means either 1900 or 2900 FORTRAN. The tables do not generally include any FORTRAN statement, syntax, or pecularity where CFT is believed to be at least as general as the other dialects shown. "Unusual" and its synonyms mean "non-CFT."

TABLE 1 HARDWARE DEPENDENCIES

Item	CFT	ANSI-66	ANSI-77	CDC	IBM	UNIVAC	ICL	HONEYWELL
Number of characters per word, bits per character Character manipulation in CFT cannot involve more then 8 characters per word. Card images require 10A8 format i.e., 10 words to store image.	8,8 .	Unspecified	Unspecified	10,6	4,8	A: 4,9 V: 6,6	2900: 4,9 1900: 6,6	
True/false representation.	Negative,	Unspecified	Unspecified	FTN: Neg, 0		True: LSB=1		True ≠ 0
Equivalences to logicals or binary settings of logicals	Positive			RUN: Pos,0		False: LSB=0		False = 0
Collation: letters before or after number, each contiguous. Testing for character sequences; letters greater than numbers internally for CFT.	After, yes	Unspecified	Unspecified	Before,Yes	Before, No	After, Yes	Before, No	
Internal Character Code	ASCII			Display Code	EBCDIC	A:ASCII (9 Bit) V:Field Data (6 Bit)		
Binary constants Z... means Z followed by a string of digits, etc.	...B	Not allowed	Not allowed	FTN...B RUN O... OR...B	Z...	O...	Z...	O...

TABLE 2 CODING FEATURES

Item	CFT	ANS1-66	ANSI-77	CDC	IBM	UNIVAC	ICL	HONEYWELL
Comment indicators (in column 1)	C,*	C	C,*	C,*,$	C	C,@ in line for rest of line	C	C,*
Continuation allowed (line length; cords)	19	19	19	19	19	1320 chars total	19	19
Multiple statements per line, separator is:	Not allowed	Not allowed	Not allowed	$	Not allowed	@ for comment	Not allowed	;
Multiple replacement statement	Not allowed	Not allowed	Not allowed	Yes	Not allowed	Not allowed	1900; Yes 2900; No	Not allowed
PROGRAM statement	Defines program name		Defines program name	Defines files			Name only allowed	
Pseudo functions (functions usable on left or right of =); must be recoded without pseudo functions.	Not allowed	Not allowed	Not allowed	Not allowed	Not allowed	BITS SUBSTR	Not allowed	
Partially reserved words some names may be illegal; O...means names beginning with O.	FORMAT END	None	END	FTN: FORMAT FUNCTION RUN: CALL,END O...,etc.		O...	Z...	
Unusual characters allowed	None	None	None	.	& in CALL ' in MS I/O $ in name	& in CALL and for concatenation ' in MS I/O $ in name	& in CALL	‡ for ** & for continuation ' in MS I/O ; for statement terminator

TABLE 3 DECLARATIVES AND ORDERING

Item	CFT	ANSI-66	ANSI-77	CDC	IBM	UNIVAC	ICL	HONEYWELL
Data assignable in declarative	No	No	No	No	Yes	Yas	Yes	Yas
Precisions of variables (by *n) in type statement	Yas	Not allowed	Not allowed	Not allowed	Yas		Yas	
"TYPE" type allowed	No	No	No	Yas	No	Yes		Yes
Non CFT types	None	None	Character	ECS		Character	Character	Character Abnormal
DATA(...) allowed	No	No	No	Yes	No	No	No	
Arithmetic statement functions can be other than after declarative before executable statements	No	Not allowed	No	FTN: No RUN: Yas				
Parenthesis required in PARAMETER	Yas	Not allowed	Yas	Yes	Yas	No	Yas	No
IMPLICIT must precede all other declaratives, executables	Yas	Not allowed	Yas	Yas	Yas	No	Yes	No
Results of DATA statement type mismatch. For RUN and UNIVAC FORTRAN V must correct types of constants in DATA statements.	Converted except for logical, complex	Not allowed	Converted except for logical character	RUN: ignored		A: Convert V: Ignore		
COMMON irregularities; Initial common block lengths must be as long as ever required. Numbered common must be changed to named.	None	None	None	Numbered COMMON blocks changing lengths of COMMON	None	None	None	

A

TABLE 4 NAMES OF VARIABLES

Item	CFT	ANSI-66	ANSI-77	CDC	IBM	UNIVAC	ICL	HONEYWELL
Maximum number of characters in names	8	6	6	7	6	6	32	8
$ in name	Not allowed	Not allowed	Not allowed	Not allowed	Yes	After initial latter	Not allowed	Not allowed
Lower case characters	Not allowed	Not allowed	Not allowed	Not allowed	Not allowed	Treated as upper case	Not allowed	Treated as upper case

TABLE 5 CONSTANTS, LITERALS, AND STRINGS

Item	CFT	ANSI-66	ANSI-77	CDC	IBM	UNIVAC	ICL	HONEYWELL
Non CFT types Quad precision change to double and double to single, probably.	None	None	Character	Not allowed	Quad precision double-complex	Quad precision double-complex character	Quadruple double-comp. character.	Character
Nonstandard length identifiers (such as READ*D for REAL*8) Double precision and Quad precision functions must be changed to correspond to actual argument types where these are changed.	None	None	None	None	None	Not allowed	I,J,K,L,M,N, E,R,Q,D.	
Unusual constant forms	...B for octal	Not allowed	Not allowed	FTN: ...B RUN: ...B or O... for octal	Z... for Hex Data init- ialization only.	O... for octal	1900:O... for octal 2900:Z... for Hex	O... for octal
Alternate character codes (See also third hardware item)	None	Not Specified	Not Specified	Display code	EBCDIC	V: Field data		

TABLE 6 ARITHMETIC AND EXPRESSIONS

Item	CFT	ANSI-66	ANSI-77	CDC	IBM	UNIVAC	ICL	HONEYWELL
I/J/X = FLOAT(I/J)/X For CDC Subscript (I) can be added in arithmetic statements for subscripted variables if one is not present.	Yes	Not allowed	Yes	No	Yes	Yes	Yes	
Is A**-B ok (and other similar ones)	Not allowed	Not allowed	Not allowed	Not allowed	Not allowed	A: No V: Yes	Not allowed	
Subscripts required for multi-dimensional arrays	At least one	All	All	None	All	All		
Non-integral subscripts	Not allowed	Not allowed	Not allowed	FTN: Yes RUN: No	Yes; real	Yes	Yes; real only	

TABLE 7 BRANCHING AND CONTROL STATEMENTS

Item	CFT	ANSI-66	ANSI-77	CDC	IBM	UNIVAC	ICL	HONEYWELL
Results of out-of-range computed GO TO.	Fall through	Not allowed	Fall through	Fatal error	Fall through	Fall through	Fall through	Fatal error
Arithmetic IF can have missing statement labels.	Not allowed	Not allowed	Not allowed			Yes, GOTO can also	1900: Yes 2900: No	Yes
Arithmetic IF can have complex argument	Not allowed	Not allowed	Not allowed	Yes; only real part tested	Not allowed	Not allowed		
DO 1 I = 10,1 executes This is a difficult error to monitor unless IF's are inserted for all DO statements but it can be fixed with ON=J on the CFT statement	No times	Not allowed	No times	Once	Once	As if DO 1 I=10,1,-1	Once	
Complex relational operator. Complex	.NE. and .EQ. only	Not allowed	.NE. and .EQ only	OK; only real part tested except .NE. and .EQ.	Not allowed			
Labels on non-executable non-FORMAT statements	Not allowed	Not allowed	Labels allowed; Reference not allowed		Not allowed	Not allowed can and DO. V: Yes	A: FORMAT	Not allowed

TABLE 8 INPUT/OUTPUT/FORMATTING

Item	CFT	ANSI-66	ANSI-77	CDC	IBM	UNIVAC	ICL	HONEYWELL
Free format I/O	Not allowed	Not allowed	* for statement label in I/O statement	* for statement label	* for atatement label	* for statement label	* for statement label	Not allowed
End-of-file, error checks	END=, ERR=, EOF, UNIT, IEOF		END=, ERR=, IOSTAT=	FTN: EOF IOCHEX RUN: EOF,U IOCHEX,U	END=, ERR=	END=, ERR=		END=, ERR=, IOSTAT=
TAPE, I/O TAPE, etc. allowed with R/W atatements	Not allowed	Not allowed	Not allowed	FTN: No RUN: Yes	Not allowed	A: No V: Yea	Not allowed	
Random mass storage I/O	GETPOS SETPOS	None	READ(..REC=) WRITE(..NEC=)	READMS/ WRITEMS etc.	FIND (a'r) etc.	FIND (a'r) etc.	READ(u,fl,clause) list, etc.	Like IBM
Other I/O statements	Not allowed	Not allowed	OPEN CLOSE INQUIRE	READEC WRITEC MOVLEV	READ(U,ID=w) AD(UV) DEFINF, etc.			
None CFT FORMAT specs	None	None	Ew.dEe,etc.	Ew.dEe, etc.	Qw.d E,F ok for integers; G ok integer, logical	Iw.d, Jw, +S Ew.dEe, etc.	Qw.d, V, Mw.c G ok for logical or integer	
Format paren nesting maximum	3			FTN: 3 RUN: 2	3	5+		
Encode/Decode pecularities	None	Not allowed	READ & WRITE to character strings		REREAD	Char count optional; no. of chars converted available. ERR= allowed		Char count not included. ERR= allowed.

TABLE 9 SUBROUTINES AND FUNCTIONS

Item	CFT	ANSI-66	ANSI-77	CDC	IBM	UNIVAC	ICL	HONEYWELL
In program: RETURN=STOP In subroutine: END = RETURN	No Yes	No Yes	No Yes	FTN: Yes/Yes RUN: Yes/No	Yes No	"Internal" subroutines allowed	STOP required	
Alternate returns syntax	* before label in CALL * in subr.	Not allowed	* before label in CALL * in subr.	(-,-,...) RETURNS(-,-,...)	& before label in CALL & in subr.	A:& or $ before label in CALL. V: Indexing is absolute, not relative.	& before label in CALL	FORTRAN 77 syntax used
ENTRY has own calling sequence and usable as a function CDC ENTRY statements must have correct calling sequences added.	Yes	Not allowed	Yes	No	Yes	Yes; same type as function	Yes	Yes
END statement required	Yes		Yes	Yes		No	Yes	Yes
Dummy argument in slashes = call by address	Illegal	Illegal	Illegal	Illegal	Yes	Yes	Yes	Illegal
"& subprogram name" allowed	Not allowed	Not allowed	Not allowed	Not allowed	H: Yes G: No	In EXTERNAL only	Not allowed	Not allowed
Overlay syntax	LDR commands ROOT, POVL, SOVL CALL overlay (...)	Unspecified	Unspecified	OVERLAY(...)		Use BANK statement		
DEFINE used in arithmetic statement functions	Not allowed	Not allowed	Not allowed	Not allowed	Allowed	Allowed	Not allowed	

A

TABLE 10 INTRINSIC OR INLINE FUNCTIONS

Item	CFT	ANSI-66	ANSI-77	CDC	IBM	UNIVAC	ICL	HONEYWELL
Memory management	None	None		ECS	None	BANK		
I/O		Not allowed		Unload	Unload			
Compiler directives	CDIR$: LIST NOLIST, EJECT etc.	Not specified		FTN: No RUN: between routines, . in column 1	Generic	Compiler		
Assembler code with FORTRAN	Not allowed	Not allowed	Not allowed	IDENT	Not allowed	Not allowed		
Code insertion	Via UPDATE	Not allowed		Via UPDATE		Include Delete		
Actual functions that are unusual	None	None	None	SHIFT RUN: FORTRAN II syntax	All double precision Q prefixes on functions	All double precision	Double precision + D,E,Q,R,I,J,K prefixes; fatal functions; many additional such as trig with degrees.	

TABLE 11 EXTERNAL FUNCTIONS

Item	CFT	ANSI-66	ANSI-77	CDC	IBM	UNIVAC	ICL	HONEYWELL
General differences	None	None	None		All double precision	All double precision	All double precision	
Specific functions				FORTRAN II functions: LOCF,ABSF, etc SLITE DISPLA TIME IOCHEC		GAMMA LGAMMA	Time, date	

A

APPENDIX C

TRACEBACK

There is a FORTRAN library routine TRBK (file) that you can call to determine how the program reached the current routine. If there is no argument, the trace is printed in the logfile, if file is specified, '$OUT' perhaps, the traceback goes to that file.

When a task terminates abnormally, TRBK is automatically called so a trace of the subroutine calling tree to the point of error is provided. To make this as useful as possible, turn on the block listing from CFT:

          CFT,ON=B, ...

and the load map on the loader card: LDR,MAP,... so that the addresses provided can be easily localized in the FORTRAN source. Below is an annotated abort message to illustrate using the trace information.

     AB053 - FLOATING POINT ERROR

     AB000 - JOB STEP ABORTED.   P = 0144760A

          - ******** WAS CALLED BY SOLVE   AT LOCATION 0144760A

          - SOLVE   WAS CALLED BY EQNS   AT LOCATION 0012341C

          - EQNS   WAS CALLED BY $MAIN   AT LOCATION 0003411A

The cause of termination is a floating point overflow from the first line of the abort message. Another common diagnostic is "OPERAND RANGE ERROR", which occurs when an attempt is made to reference some part of memory outside your user area, most likely a wild index. If all output is lost, one possible cause is using block common with a DIMENSION of 1, COMMOM X(1), but storing into Xs with subscripts much larger than 1. (This may overwrite I/O buffers and tables.)

The last instruction being executed at the time of abort was at location 00144760A. For the FORTRAN programmer, discard the parcel, A in this case, leaving the OCTAL address of the instruction word. The actual error probably occurred on earlier instruction. The load map ADDRESS gives the base address of all routines. In this case, find the base address of SOLVE, where the error occurred (from the third line of the abort message). Subtract the base address from the absolute P-counter address given to find the relative address in SOLVE. (Remember to subtract in OCTAL!!!)

Because the BLOCK (ON=B) listing was turned on for CFT, you will have a list of all blocks which will allow you to find the FORTRAN code block where the abort occurred:

P-Counter   0144760 (A)
- Loader ADDRESS of SOLVE   -   0104700

Relative address in SOLVE   0040060

Partial block listing for SOLVE:

.
.
.

SOLVE VECTOR BLOCK BEGINS AT SEQ. NO. 1372, P=40035B
SOLVE BLOCK BEGINS AT SEQ. NO. 1380, P=40055D
SOLVE BLOCK BEGINS AT SEQ NO. 1401, P=40077A

.
.
.

Because the relative error address is 40060, which is between 40055 and 40077. The bomb occurred for a FORTRAN statement between numbers 1380 and 1401. The listing should now help you find the probable source of the error quickly.

Similarly, the point at which SOLVE was called by EQNS can be determined. The absolute address of the CALL was 0012341C and EQNS was called by the main program at absolute locations 0003411A

APPENDIX D

COMPILER DIRECTIVES

Compiler directive lines begin with characters CDIR$ in columns 1
through 5 and any of the directives listed below in columns 7 through 72.

DIRECTIVE	FUNCTION
EJECT	Ejects to top of next page.
LIST	Resumes listable output.
NOLIST	Suppresses production of listable output.
CODE	Produces code list.
NOCODE	Suppresses production of CFT-generated code lists.
VECTOR	Enables vectorization of inner DO-loops.
NOVECTOR	Suppresses vectorization of inner DO-loops.
IVDEP	Ignores vector dependencies in the next DO-loops.
INT24	Identifies listed variables and arrays as 24-bit integers, equivalent to INTEGER *2 declarative.
FLOW	Enables flowtrace.
NOFLOW	Disables flowtrace.
SHED	Enables the scheduler.
NOSCH	Disables the scheduler.
VFUNCTION	Identifies external vector functions.
BOUNDS	Checks array references for out-of-hand subscripts.

APPENDIX E

CHARACTER SETS

CHAR	ASCII	HEX	ASCII CARD CODE	CHAR	ASCII	HEX	ASCII CARD CODE
NUL	000	00	12-0-9-8-1	@	100	40	8-4
SOH	001	01	12-9-1	A	101	41	12-1
STX	002	02	12-9-2	B	102	42	12-2
ETX	003	03	12-9-3	C	103	43	12-3
EOT	004	04	9-7	D	104	44	12-4
ENQ	005	05	0-9-8-5	E	105	45	12-5
ACK	006	06	0-9-8-6	F	106	46	12-6
BEL	007	07	0-9-8-7	G	107	47	12-7
BS	010	08	11-9-6	H	110	48	12-8
HT	011	09	12-9-5	I	111	49	12-9
LF	012	0A	0-9-5	J	112	4A	11-1
VT	013	0B	12-9-8-3	K	113	4B	11-2
FF	014	0C	12-9-8-4	L	114	4C	11-3
CR	015	0D	12-9-8-5	M	115	4D	11-4
SO	016	0E	12-9-8-6	N	116	4E	11-5
SI	017	0F	12-9-8-7	O	117	4F	11-6
DLE	020	10	12-11-9-8-1	P	120	50	11-7
DC1	021	11	11-9-1	Q	121	51	11-8
DC2	022	12	11-9-2	R	122	52	11-9
DC3	023	13	11-9-3	S	123	53	0-2
DC4	024	14	4-8-9	T	124	54	0-3
NAK	025	15	9-8-5	U	125	55	0-4
SYN	026	16	9-2	V	126	56	0-5
ETB	027	17	0-9-6	W	127	57	0-6
CAN	030	18	11-9-8	X	130	58	0-7
EM	031	19	11-9-8-1	Y	131	59	0-8
SUB	032	1A	9-8-7	Z	132	5A	0-9
ESC	033	1B	0-9-7	[	133	5B	12-8-2
FS	034	1C	11-9-8-4	\	134	5C	0-8-2
GS	035	1D	11-9-8-5	]	135	5D	11-8-2
RS	036	1E	11-9-8-6	^	136	5E	11-8-7
US	037	1F	11-9-8-7	_	137	5F	0-8-5
Space	040	20	None	`	140	60	8-1

CHAR	ASCII	HEX	ASCII CARD CODE	CHAR	ASCII	HEX	ASCII CARD CODE
!	041	21	12-8-7	a	141	61	12-0-1
"	042	22	8-7	b	142	62	12-0-2
#	043	23	8-3	c	143	63	12-0-3
$	044	24	11-8-3	d	144	64	12-0-4
%	045	25	0-8-4	e	145	65	12-0-5
&	046	26	12	f	146	66	12-0-6
'	047	27	8-5	g	147	67	12-0-7
(	050	28	12-8-5	h	150	68	12-0-8
)	051	29	11-8-5	i	151	69	12-0-9
*	052	2A	11-8-4	j	152	6A	12-11-1
+	053	2B	12-8-6	k	153	6B	12-11-2
,	054	2C	0-8-3	l	154	6C	12-11-3
-	055	2D	11	m	155	6D	12-11-4
.	056	2E	12-8-3	n	156	6E	12-11-5
/	057	2F	0-1	o	157	6F	12-11-6
0	060	30	0	p	160	70	12-11-7
1	061	31	1	q	161	71	12-11-8
2	062	32	2	r	162	72	12-11-9
3	063	33	3	s	163	73	11-0-2
4	064	34	4	t	164	74	11-0-3
5	065	35	5	u	165	75	11-0-4
6	066	36	6	v	166	76	11-0-5
7	067	37	7	w	167	77	11-0-6
8	070	38	8	x	170	78	11-0-7
9	071	39	9	y	171	79	11-0-8
:	072	3A	8-2	z	172	7A	11-0-9
;	073	3B	11-8-6	{	173	7B	12-0
<	074	3C	12-8-4	|	174	7C	12-11
=	075	3D	8-6	}	175	7D	11-0
>	076	3E	0-8-6	~	176	7E	11-0-1
?	077	3F	0-8-7	DEL	177	7F	12-9-7